An Autonomic Testing Framework for IPv6 Configuration Protocols

Sheila Becker[1], Humberto Abdelnur[2], Radu State[1], and Thomas Engel[1]

[1] University of Luxembourg, Luxembourg
{sheila.becker, radu.state, thomas.engel}@uni.lu
[2] MADYNES - INRIA Nancy-Grand Est, France
humberto.abdelnur@loria.fr

Abstract. The current underutilization of IPv6 enabled services makes accesses to them very attractive because of higher availability and better response time, like the IPv6 specific services from Google and Youtube have recently got a lot of requests. In this paper, we describe a fuzzing framework for IPv6 protocols. Fuzzing is a process by which faults are injected in order to find vulnerabilities in implementations. Our paper describes a machine learning approach, that leverages reinforcement based fuzzing method. We describe a reinforcement learning algorithm to allow the framework to autonomically learn the best fuzzing mechanisms and to automatically test stability and reliability of IPv6.
Keywords: Fuzzing, IPv6, Reinforcement Learning

1 Introduction

The prediction that IPv6 implementations will increase due to the limited address space of IPv4 persists since a long time. Actually, a second reason for IPv6 to sprout, occurs because of the current underutilization of IPv6 enabled services, as services provided from Google and Youtube, makes accesses to them very attractive due to the better response time and higher availability, and these services receive a lot of requests. Although the usage of IPv6 grows, appropriate testing and fuzzing frameworks are lacking. These frameworks are important to assure the reliability of network protocols. For this reason, we assess a fuzzing framework for IPv6 protocols in order to detect vulnerabilities with the objective to appraise reliability.

In this paper, we start our analysis by defining a behavioral model of IPv6 protocols with a Finite State Machine of the given protocol. Afterwards, we inspect the different message types exchanged within the protocol by defining for every field, a message is composed of, the type of the field, the default value, and the length of the field. Subsequent to this inspection, we investigate on possible fuzzing strategies to guarantee efficiency while fuzzing. We employ these fuzzing strategies in an IPv6-implementation and we propose the usage of two different reward functions, one based on what is sent to the network, and the other based on kernel tracing. Finally, we use these reward functions to lead a reinforcement learning algorithm for learning the fuzzing strategy with the highest reward.

The paper is structured as follows: In Section 2 we explain fuzzing and the different existing fuzzing mechanisms. We introduce reinforcement learning and explain which model we adopt within our framework in Section 3. Section 4 introduces the analyzed IPv6 protocol and how this analysis is used for implementing the different fuzzing strategies, and it explains how to integrate reinforcement learning in our fuzzing framework to make it intelligent. In Section 5 we discuss related work. Lastly, we conclude our work and determine future work in Section 6.

2 Fuzzing

Fuzzing is known as a special case of software testing. It is a method to discover faults by providing unexpected input while exceptions are monitored. Basically, there are two different fuzzing styles. On one side, there is the generation based fuzzer, where the input is created from scratch. While on the other side, the mutation based fuzzer mutates already existing input but in this case the input needs to be captured beforehand, i.e. with the help of wireshark[3]. The intention of fuzzing is to expose vulnerabilities of an application, or as in our case, a network protocol. So, it allows to uncover format string vulnerabilities, buffer-overflows, and integer-overflows.

One example of such a vulnerability is shown in the following code of the Solaris 8 IP stack [5]

```
uint8_t
ipoptp_first(ipoptp_t *optp, ipha_t *ipha){
        uint32_t totallen;
        totallen=
          ipha->ipha_version_and_hdr_length
          -(uint8_t)((IP_VERSION<<4)+
          IP_SIMPLE_HDR_LENGTH_IN_WORDS);
        totallen<<=2;
        ...
}
```

The first two fields of the IP header are treated as one field with two components, the code assumes that the size of the IP options comes out of subtracting a static value from the first byte. This can lead to a kernel crash when the IP version is less than 4.

The first approach for fuzzing [14] was to see the whole input as a sequence of bits and to randomly flip the bits, this is called random fuzzing. Random fuzzers do not have any knowledge of the protocol, and the input space can be tremendous. Therefore, it is more effective to use block-based fuzzers [2]. This way, a message is divided in different blocks, the fixed strings and the variable values. Only the variable input will be fuzzed as it makes no sense to fuzz the fixed strings, as usually the fixed strings are not considered for processing

[3] http://www.wireshark.org/

a message. The only thing that may have an impact is to change the size of the fixed fields. In comparison to random fuzzing, block-based fuzzing has the advantage, that we have knowledge of the type of the fields, this way, we squeeze the input space not only by ignoring the fixed strings but also by restricting the fuzzing of some particular fields. Some of these particular fields e.g. the checksum field, are inspected before the messages are being processed, and in case the field is obviously erroneous, the messages will not be processed but dismissed. Concluding, it is useless to fuzz the checksum field as we do not want the message to be deleted even before penetrating the implementation.

3 Reinforcement Learning

Reinforcement Learning [10], [18], is known as the problem of an agent that needs to learn by trial-and-error interactions with a dynamic environment. The agent gets a reward or punishment based on the interaction taken. In a reinforcement learning model we have a set of states an agent can be in, and a set of actions for each state. The agent is in one state and has to choose an action based on input that provides indications on the current state. Thereafter, the agent swaps to a different state, and has to choose another action. After each swap the agent receives a reward, or in some models even a punishment. The objective in reinforcement learning is to optimize the reward in long-term.

Different algorithms and models have been introduced in order to solve reinforcement learning problems as for temporal difference learning there are Q-Learning [10], [18] , or the SARSA algorithm [18]. Where temporal-difference (TD) learning combines Monte Carlo ideas and dynamic programming ideas. TD incorporates the advantages of both ideas, it learns immediately from raw experience without a model of the dynamics of the environment. Estimates are updated based on previous experience. Q-learning is an Off-Policy TD control. The action-value function Q directly approximates the optimal action-value function Q^* without following a behavior policy. Nevertheless, the policy is followed for determining visited and updated state-action pairs.

SARSA stands for State-Action Reward State-Action and is an On-Policy TD control. In other reinforcement learning models, only transitions between states are appraised. In the SARSA algorithm transitions from state-action pair to state-action pair are considered. This algorithm is an on-policy TD control as it follows a behavior policy Π. Action values are evaluated and estimated for this behavior policy, which must be adjusted towards an optimal policy so that the algorithm performs optimal. We focus on the SARSA algorithm for our work, as it is an online temporal-difference learning for transitions between state-action pairs. The following equation shows how the values are updated after each episode.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{1}$$

$$\Pi_t(s,a) = Pr\{a_t = a | s_t = s\} = \frac{e^{Q(s,a)}}{\sum_b e^{Q(s,b)}} \quad (2)$$

Equation 1 defines how the estimated values for action a in state s are updated. $Q(s_t, a_t)$ represents the value of performing the action a_t while being in state s_t in moment t. We assume that we have a finite number of states S and a finite set of actions A. The reward is represented by r, α is a step-size parameter, and γ is a discount rate to determine the importance of future events, is $\gamma \to 0$ then it is considered to be shortsighted, is $\gamma \to 1$ then it is considered as being farsighted. We also assume to have an action selection probability $\Pi_t(s,a)$ that assigns for every state probabilities for performing each action as defined in Equation 2. The objective of reinforcement learning is to learn appropriate values for the function Q and to estimate the right probabilities Π_t. Basically, Q measures the efficiency of each action in each state while Π_t is useful for driving the exploration path. We see from Equation 1 that actions that have resulted in positive feedbacks will increase the corresponding Q-values and will make these actions more probable in the future (by increasing the probability of these actions).

4 IPv6 Protocol Analysis & Fuzzing Framework

This section describes a reinforcement driven fuzzing framework for IPv6. We consider the Neighbor Discovery Protocol, a simple but heavily used protocol in IPv6, to illustrate our case.

4.1 Neighbor Discovery Protocol Analysis

The Neighbor Discovery (ND) Protocol replaces the Address Resolution Protocol (ARP), and the ICMP Router Discovery and Redirect from IPv4. It is used amongst others for address auto-configuration, to get knowledge about network prefixes, routes, configuration information, link-layer addresses, and to detect duplicate IP addresses. This protocol is composed of five different message types:

- Neighbor Solicitation
- Neighbor Advertisement
- Router Solicitation
- Router Advertisement
- Redirect Message

Behavioral model

Finite State Machines (FSM) can be used to model the behavior of protocols. An FSM consists of states and transitions between states. An FSM helps to see what messages are sent and received in which state. In Figure 1 we can see the defined FSM, where *NS* stands for Neighbor Solicitation, *NA* for Neighbor Advertisement, *RS* for Router Solicitation, and *RA* for Router Advertisement.

The states *S1* to *S7* describe the address-auto configuration that is launched when a node is new in a network, as well as for Duplicate Address Detection. For address-auto configuration in ND, the new node sends out a Neighbor Solicitation with its own address, if it receives no Neighbor Advertisement, it knows that the chosen address is not used, so it can use that address. Thereafter, it sends a Router Solicitation message, in order to configure itself correctly according to the information it receives from the Router Advertisement. In case it does not receive a Router Advertisement, it will make a default configuration. Once, the configurations are finished it is part of the network and it will periodically use the previous method in order to detect potential duplicate address in the network.

The states from *S7* to *S11* represent the normal behavior of a configured node, that sends Router Solicitations and Neighbor Solicitations. The redirect message is not present in this FSM as this type of message is sent by the router to inform a node, that a better route exists. In the defined FSM we focussed only on the behavior of a node. The behavior of the router is not taken into account in this work, as it would have only little impact on our work.

In order to have a complete view of a protocol, to model an FSM is not sufficient. We also need to analyze message formats.

Message Inspection

In order to analyze message formats, we decompose the messages into the fields they are composed of. Once we have all the fields of one message, we define what type each field has, and what the default value is of that field as shown in Table 1.

Table 1. Decomposition of the Neighbor Advertisement message

Neighbor Adverstisement	
<IPsource><IPdest><HopLimit><ICMP-Fields >	
<IPsource>	(bin, 128bits, prefix + interface address)
<IPdest>	(bin, 128bits, multicast address)
<HopLimit>	(HopLimit, 8bits, 255)
<ICMP-field1>	(type, 8bits, 136)
<ICMP-field2>	(code, 8bits, 0)
<ICMP-field3>	(checksum, 16bits, ICMP checksum)
<ICMP-field4>	(R, router flag, 1bit)
<ICMP-field5>	(S, solicitation flag, 1bit)
<ICMP-field6>	(O, override flag, 1bit)
<ICMP-field7>	(reserved, 29bits, unused)
<ICMP-field8>	(target address, bin, 128bits)

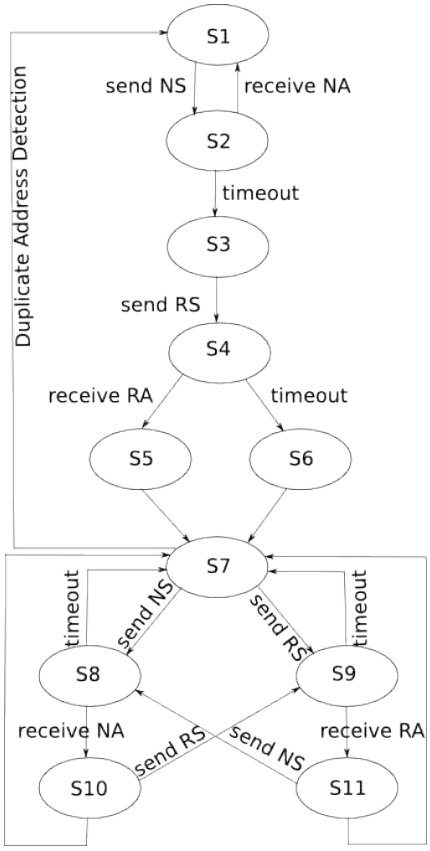

Fig. 1. Finite State Machine of the Neighbor Discovery Protocol

4.2 Fuzzing Strategies

After analyzing the protocol and the messages, we need to define what strategies we want to apply in order to detect vulnerabilities efficiently. Many different methods and strategies exist for fuzzing [9].

1. Deleting fields
2. Inserting fields
3. Modifying value of a field
4. Inserting message based on the behavioral model
5. Repeating message based on the behavioral model
6. Dropping message based on the behavioral model

One possibility is to modify one or more data fields. This will have as consequence that an endpoint in the network has to handle wrong input. The data fields can be modified by deleting the fields, by inserting fields, or by modifying the value of the data field. Another possibility is to change the message type, by saying

Table 2. Fuzzing of the Neighbor Advertisement message

Neighbor Adverstisement	
<IPsource><IPdest><HopLimit><ICMP-Fields >	
<IPsource>	(bin, 128bits, prefix + interface address)
<IPdest>	(bin, 128bits, multicast address)
<HopLimit>	(HopLimit, 8bits, 255)
<ICMP-field1>	**(type, 8bits, 137) (1)**
<ICMP-field2>	(code, 8bits, 0)
<ICMP-field3>	(checksum, 16bits, ICMP checksum)
<ICMP-field3>	**(checksum, 16bits, ICMP checksum) (2)**
<ICMP-field4>	(R, router flag, 1bit)
<ICMP-field5>	**(S, Solicitation flag, 1bit) (3)**
<ICMP-field7>	(reserved, 29bits, unused)
<ICMP-field8>	(target address, bin, 128bits)

that this message is of another type. In Table 2 examples for fuzzing the neighbor advertisement message are shown. **(1)** shows the changed type field, originally the ICMP message type is 136, here we changed it to 137. **(2)** shows the duplicate checksum field. In **(3)** we deleted the ICMP-field 6.

These described methods mutate the content of the message, but we can also mutate a message-order, i.e.sending a different order of messages by inserting, repeating, or dropping messages inside one session. Practically, this means that we need to create/mutate packets with changed data fields, based on the protocol and message analysis we have done previously.

Strategical behavior

If we were to model the fuzzing strategies and the associated protocol state machine like one single state machine, we would need one state for each combination of fuzzing strategy and state. This can lead to a state explosion because we have for each state six fuzzing strategies that we applied to k fields. Resulting, we have k^6 possibilities. For our example, considering the Neighbor Advertisement message where we have 11 fields, this means that we have 11^6 possibilities for only one state and concluding, we have $11 * 11^6$ possibilities as we have 11 states in our FSM. This leads to a state space explosion.

To counteract this problem, we determine another FSM but not based on the behavioral model of the protocol but on the strategical model, meaning that we will use the different fuzzing strategies as state model as shown in Figure 2. The states are mapped to the fuzzing strategies as follows:

- A - Deleting fields
- B - Inserting fields
- C - Modifying value of a field
- D - Inserting message based on the behavioral model
- E - Repeating message based on the behavioral model
- F - Dropping message based on the behavioral model

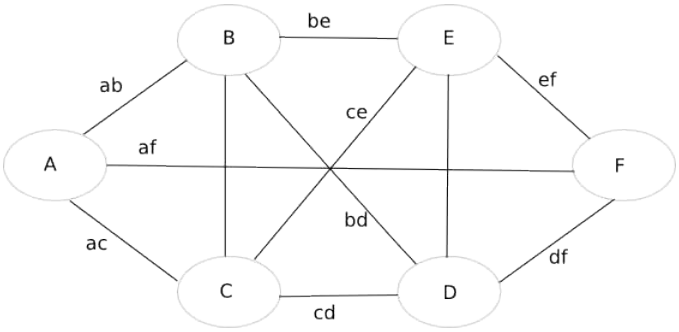

Fig. 2. Fuzzing strategies as behavior model

This model has two advantages in comparison to the usage of the FSM of the behavioral model of the protocol. On one side we counteract the state space explosion, and on the other side we provide a general fuzzing model, that can be adapted to other network protocols.

4.3 Framework for adopting a Reinforcement Learning model

For adopting a reinforcement Learning model, we need to have a reward function, so that we can deduce for every action in a specified state a reward. For a reward function, one needs to quantify the impact of IPv6 messages sent to one machine, or to a whole network. One way to see and quantify the impact on the host, lies in pursuing a message and to see the amount of function or system calls executed due to that message. Therefore, we need to trace the IPv6 messages. For this purpose we configure the kernel so that we can trace and follow the mutated IPv6 messages. We start by downloading a kernel compilation via git-core. We configure the kernel with the command *make menuconfig*, under kernel hacking we can find the section *tracers* where we can enable *kernel function tracer*. This way we can trace function calls, as *memcpy*, *kmalloc*, etc.

Furthermore, we can debug the IPv6 modules of the kernel, by integrating debugging code. After the recompilation of the kernel, we have output of the protocol implementation and we see what code has been executed in which module. We also have information if errors occurred while processing the message. Finally, we can take this information and allocate the number of methods or functions called to the fuzzed messages. Based on this, we have a quantifying mechanism for our reward function. Another way to quantify the impact of our fuzzing framework, is to monitor the messages sent from the host to be fuzzed. This way we can see if the host responds correctly or not, and if the responses are delayed.

Hence, we can conclude three different reward functions for our framework, one for the tracing, one for the debugging, and another one for monitoring the network. The first reward function is based on the entropy and the power the

function calls produce due to the fuzzed input. The entropy represents the heterogeneity of the fuzzing framework as the distribution of the functions called due to one message over all available functions where q_i is the number of different functions called. The entropy of a message q_t is defined as:

$$H(q_t) = -\sum_{i=1}^{m} p_{t,i} log(p_{t,i}) \tag{3}$$

where:

$$p_{t,i} = \frac{q_{t,i}}{\sum_{i=1}^{m} q_{t,i}}$$

The power represents the amount of the functions called due to one input message. The power is defined as follows:

$$Power(q_t) = \sqrt{\sum_{i=1}^{m} q_{t,i}^2} \tag{4}$$

We normalize these two metrics and assemble both functions, for entropy Equation 3 and for power Equation 4, to glean a resulting reward function for tracing as follows:

$$r(q_t)_{trace} = H(q_t)_{norm} + Power(q_t)_{norm} \tag{5}$$

For debugging we define following reward function:

$$r(q_t)_{deb} = \begin{cases} 1, & \text{if error monitored} \\ 0, & \text{if no error monitored} \end{cases} \tag{6}$$

For monitoring messages on the network we specify the following reward function:

$$r(q_t)_{mon} = \begin{cases} 1, & \text{if corrupt or delayed message monitored} \\ 0, & \text{if correct response and no delay monitored} \end{cases} \tag{7}$$

Finally, we can calculate an overall payoff function out of Equation 5, Equation 6, and Equation 7 as follows:

$$r(q_t) = r(q_t)_{trace} + r(q_t)_{deb} + r(q_t)_{mon} \tag{8}$$

A potential reinforcement learning process chain could look like Figure 3 shows.

The objective of this framework is to create an autonomic fuzzer, that can send wrong/fuzzed messages to a host, and that based on monitoring for exceptions, considering the responses of the host that might be corrupt or delayed, learns the impact of the fuzzed messages. While integrating reinforcement learning, we allow the fuzzer to become autonomic as it will choose the fuzzing strategy for given messages that return the optimal reward and based on the messages returned it can detect in which state it resides. This way, the framework learns the best strategies.

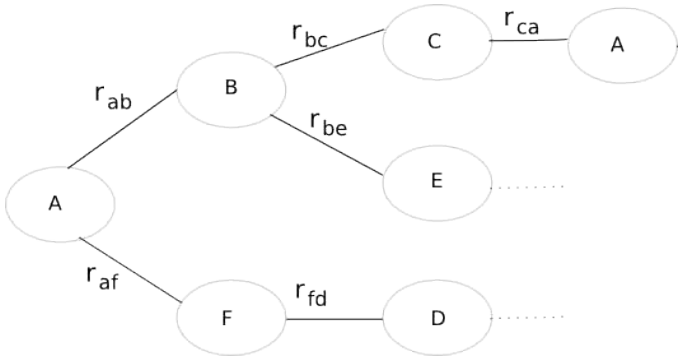

Fig. 3. Potential reinforcement learning process chain of our framework

5 Related Work

One can approach the fuzzing problem by different techniques, as using symbolic execution [3], [13], [8], where input variables are made symbolic, and constraints are assembled to these variables along the execution path. Tracing tainted data is another approach where system functions and system calls from input data are followed to trace the behavior of an application as described in [19], [16], [6]. Taint tracing has an important benefit to the symbolic execution. In taint tracing the source code of the application is not needed, whereas it is needed for symbolic execution.

Modeling techniques, as event driven models based on extended finite state machines [12], or markov models [17], are used in some papers for detecting flaws and for testing. In [9] a model-based approach for flaw detection based on a finite state machine model as well as fuzzing strategies are proposed. Reverse engineering is used in [4] for fuzzing purposes.

The authors of [11] provide a quantification of the extent of IPv6 deployment, they collect and analyze a variety of data to characterize IPv6 penetration. TTCN is a testing framework, which is used for IPv6 testing in [21] and in [20] it is used for testing Neighbor Discovery Protocol. The problem with testing is that, a system or in this case a network protocol is testing with correct input to see if the protocol works as it is specified in the RFC. Testing does not provide any framework for unexpected input, whereas this is the objective of Fuzzing.

Reinforcement learning has been addressed in many papers and books, so Sutton and Barto [18] give a good overview of the existing approaches for reinforcement learning. In [10], the major algorithms are explained. The authors of [7] propose to use graph kernels and gaussian processes for relational reinforcement learning.

To the best of our knowledge, no previous work has been investigating on Fuzzing using Reinforcement Learning methods.

6 Conclusion and Future Work

In this work, we proposed and assessed a autonomic fuzzing framework for IPv6 protocols. An overview has been given on fuzzing methods and reinforcement learning algorithms. We analyzed the Neighbor Discovery Protocol by specifying a finite state machine and decomposing the different message types. We showed how fuzzing can be used for vulnerability detection in IPv6 protocols by applying different fuzzing strategies. We adopt a reinforcement learning algorithm by defining a reward function that is composed of three different functions based on different monitoring techniques. We used reinforcement learning with the objective to make our fuzzing framework intelligent so that it will learn which fuzzing strategies are most effective.

For future work, we plan to give a proof of concept of this work and we want to apply different reinforcement learning models to see which performs best within our framework. This framework can be adopted for other network protocols, for instance SIP (Session Initiation Protocol). The future work consists in adopting this framework for other IPv6 protocols, as ICMPv6 and also for other network protocols. We plan to integrate this framework into the fuzzing framework KiF [1]

References

1. H. J. Abdelnur, R. State, and O. Festor. KiF: a stateful SIP fuzzer. In *IPTComm '07: Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications*, pages 47–56, New York, USA, 2007. ACM.
2. D. Aitel. The Advantages of Block-Based Protocol Analysis for Security Testing. Immunity Inc, `http://www.immunitysec.com/resources-papers.shtml`, February 2002.
3. Cristian Cadar, Paul Twohey, Vijay Ganesh, and Dawson Engler. EXE: Automatically Generating Inputs of Death Using Symbolic Execution. In *In Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, Virginia, USA, November 2006.
4. Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. *Security and Privacy, IEEE Symposium on*, 0:110–125, 2009.
5. Mark Dowd, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional, 2006.
6. Will Drewry and Tavis Ormandy. Flayer: exposing application internals. In *WOOT '07: Proceedings of the first USENIX workshop on Offensive Technologies*, pages 1–9, Berkeley, USA, 2007. USENIX Association.
7. Kurt Driessens, Jan Ramon, and Thomas Gärtner. Graph kernels and gaussian processes for relational reinforcement learning. *Mach. Learn.*, 64(1-3):91–119, 2006.
8. P. Godefroid, A. Kieźun, and M. Y. Levin. Grammar-based Whitebox Fuzzing. In *PLDI'2008: ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, Tucson, US, 2008.

9. Yating Hsu, Guoqiang Shu, and David Lee. A model-based approach to security flaw detection of network protocol implementations. *Network Protocols, 2008. ICNP 2008. IEEE International Conference on*, pages 114–123, October 2008.
10. Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *JOURNAL OF ARTIFICIAL INTELLIGENCE RESEARCH*, 4:237–285, 1996.
11. Elliott Karpilovsky, Alexandre Gerber, Dan Pei, Jennifer Rexford, and Aman Shaikh. Quantifying the extent of ipv6 deployment. In Sue B. Moon, Renata Teixeira, and Steve Uhlig, editors, *PAM*, volume 5448 of *Lecture Notes in Computer Science*, pages 13–22. Springer, 2009.
12. David Lee, Dongluo Chen, Ruibing Hao, Raymond E. Miller, Jianping Wu, and Xia Yin. Network protocol system monitoring: a formal approach with passive testing. *IEEE/ACM Trans. Netw.*, 14(2):424–437, 2006.
13. Rupak Majumdar and Ru-Gang Xu. Directed test generation using symbolic grammars. In *ESEC-FSE companion '07: The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 553–556, New York, NY, USA, 2007. ACM.
14. Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, 1990.
15. T. Narten, E. Nordmark, W. Simpson, and H. Soliman. Neighbor Discovery for IP version 6 (IPv6). RFC 4861 (Draft Standard), September 2007.
16. James Newsome, David Brumley, and Dawn Xiaodong Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *NDSS*, 2006.
17. S. Sparks, S. Embleton, R. Cunningham, and C. Zou. Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 477–486, Dec. 2007.
18. Richard S. Sutton and Andrew G. Barto. Reinforcement learning i: Introduction, 1998.
19. Martin Vuagnoux. Autodafé: an Act of Software Torture. In *Proceedings of the 22th Chaos Communication Congress*, pages 47–58, Berlin, 2005. Chaos Computer Club.
20. Zhiliang Wang, Xia Yin, Haibin Wang, and Jianping Wu. Automatic testing of neighbor discovery protocol based on fsm and ttcn. In *The 2004 Joint Conference of the 10th Asia-Pacific Conference on Communications and the 5th International Symposium on Multi-Dimensional Mobile Communications Proceedings.*, pages 805 – 809 vol.2, Bejing, China, 2004. IEEE Computer Society.
21. Yujun Zhang and Zhongcheng Li. Ipv6 conformance testing: Theory and practice. In *ITC '04: Proceedings of the International Test Conference on International Test Conference*, pages 719–727, Washington, DC, USA, 2004. IEEE Computer Society.